World Journal of Advanced Engineering Technology and Sciences

eISSN: 2582-8266

Cross Ref DOI: 10.30574/wjaets

Journal homepage: https://wjaets.com/

(Research Article)

Check for updates

Autonomous workload distribution for container-based micro services environments

Shamsuddeen Rabiu [1, *], Abubakar Abba [2] and Mustapha Ahmed Abubakar [3]

[1] *Department of Computer Science, Federal College of Education, Katsina, Nigeria.*
[2] *Department of Computer Science, Federal College of Education, Zaria, Nigeria.*
[3] *Department of Information Technology, Bayero University, Kano, Nigeria.*

World Journal of Advanced Engineering Technology and Sciences, 2023, 09(02), 242–250

Publication history: Received on 24 June 2023; revised on 05 August 2023; accepted on 08 August 2023

Article DOI: https://doi.org/10.30574/wjaets.2023.9.2.0226

Abstract

Microservice architecture represents a cloud application design approach that transfers the intricacies from the conventional monolithic applications to the infrastructure. It involves breaking down the application into small, containerized microservices, each responsible for a specific functional requirement. These microservices can be independently deployed, scaled, and tested through automated orchestration systems. Our paper introduces an autonomous system for distributing workloads among containerized microservices within the cloud like a swarm, designed specifically for microservices operating within OpenVZ containers. This system has the potential to enhance performance compared to existing centralized container orchestration systems.

Keywords: Docker Container; Container orchestration; Swarm algorithm; Microservices; OpenVZ Container

1. Introduction

A common issue with contemporary monolithic enterprise systems is their challenging nature to scale, comprehend, and maintain. When developed in a monolithic approach, these systems often exhibit tightly bound connections between service components and services themselves. This interdependence makes writing, understanding, testing, evolving, upgrading, and operating the system individually more complex. Furthermore, this strong coupling can result in a domino effect of failures, where a single failing service can bring down the entire system instead of isolating and dealing with the failure. Notably, prevalent application servers such as WebLogic, WebSphere, JBoss, or Tomcat often encourage this monolithic model.

Microservices-based architecture is free of these problems [1,2,3, 27,28,]. It advocates creating a system from a collection of small, isolated services, each of which owns their data, and is independently isolated, scalable and resilient to failure. Services integrate with other services in order to form a cohesive system that's far more flexible than a typical monolithic system. One of the key principles in employing a microservices-based architecture is the decomposition of the system into discrete isolated subsystems communicating over well-defined asynchronous protocols and decoupled in time (allowing concurrency) and space (allowing distribution and mobility – the ability to move services around).

While some developers and researchers view microservices as a specific implementation pattern of service-oriented architecture (SOA), the microservice pattern possesses distinct characteristics [2]. Unlike traditional SOA, microservices employ lightweight HTTP mechanisms for communication, enabling them to be independently deployable through fully automated processes. Additionally, microservices rely on minimal centralized management. In contrast to the typical Enterprise Service Bus (ESB) model used in SOA to facilitate communication between interacting software applications, microservices architecture lacks a central unit like ESB for routing. Instead, the incidental complexity that was once present within a monolithic application is now transferred to the infrastructure. This shift is made possible by

* Corresponding author: Rabiu Shamsuddeen

Copyright © 2023 Author(s) retain the copyright of this article. This article is published under the terms of the Creative Commons Attribution Liscense 4.0.

leveraging various approaches to manage the complexity, such as programmable infrastructure, infrastructure automation, and the adoption of cloud technologies [3].

Presently, we possess a significantly improved foundation for service isolation, thanks to advancements in virtualization, Linux Containers (LXC), Docker, and Unkennels [4, 29]. This progress has led to the recognition of isolation as a critical factor for achieving resilience, scalability, continuous delivery, and operational efficiency. Consequently, it has sparked growing interest in microservices-based architectures, enabling the dissection of monolithic applications and the independent development, deployment, operation, scaling, and management of services. The value of microservices and containers lies in their ability to facilitate smaller, quicker, and more frequent changes [5, 6]. While cloud computing has transformed how we handle "machines," it has not fundamentally altered the way we manage the core components. Containers, on the other hand, promise a realm that surpasses our reliance on traditional server applications and their components. One could argue that they represent the realization of the object-oriented, component-based vision for application architecture.

So, how can you construct an intelligent system using a data center comprised of simple servers? This is where tools like Google Kubernetes [7] and the open-source Apache Mesos [8] data center operating system come into play. Docker's platform, with its Machine, Swarm, and Compose tools [9], is also noteworthy in this context. The orchestration and scheduling capabilities provided by these container platforms are essential for aligning applications with the available resources. Google developed Kubernetes specifically for managing vast numbers of containers. Instead of individually assigning each container to a host machine, Kubernetes groups containers into pods. For instance, a multi-tier application, with its database in one container and application logic in another, can be organized into a single pod. This way, administrators only need to move a single pod from one computing resource to another, instead of dealing with numerous individual containers. Apache Mesos serves as a cluster manager that aids administrators in scheduling workloads on a server cluster. It excels at handling substantial workloads, such as those involving Spark or Hadoop data processing platforms.

Docker Swarm, on the other hand, functions as a clustering and scheduling tool that automatically optimizes a distributed application's infrastructure based on the application's lifecycle stage, container usage, and performance requirements. All these container orchestration systems are monolithic applications running as daemons on dedicated cloud nodes. They orchestrate containers in a centralized manner, providing effective management and coordination of containerized applications.

Decentralized orchestration systems typically lead to performance enhancements. For instance, when composite web services are orchestrated in a decentralized manner, it results in improved throughput, scalability, and reduced response time [10. This paper proposes a decentralized system for load balancing containerized microservices. Section 2 delves into the analysis of the internal workings of a virtualization container. Additionally, the container can host an extra process that implements mobile agent intelligence for cloud applications. In Section 3, the swarm-like algorithm for container migration in the cloud is introduced, where the number of containers on each host acts analogously to pheromones in insect colonies or simple transceivers mounted on autonomous robots. Section 4 presents some preliminary experimental results for a basic cloud comprising 18 hosts. The paper concludes with a summary and brief remarks in Section 5.

2. Container Anatomy

The startup time for a container is approximately one second, whereas public cloud virtual machines can take tens of seconds to several minutes due to booting a full operating system each time. As a result, the cloud industry has been shifting away from self-contained, isolated, and monolithic virtual machine images in favor of container-type virtualization [11,12]. Containers bring autonomy to applications by bundling them with the required libraries and binaries, preventing conflicts between apps that rely on essential components of the host operating system. Unlike virtual machines, containers do not include an operating system kernel, making them faster and more flexible. Container-type virtualization allows running multiple isolated sets of processes, each set designated for a specific application, all under a single kernel instance. This isolation capability enables the complete state of a container to be saved (or checkpointed) and later restarted. This feature allows for checkpointing a running container and restarting it later on the same or a different host in a transparent manner, without affecting running applications and network connections [11,13].

This paper employs container-type virtualization to construct a cloud swarm consisting of tasks. Each container not only includes the application and its libraries but also houses a separate process acting as a mobile agent [14, 15, 16]. This agent is responsible for sensing neighbouring containers and initiating the live migration of its container to another host. Compared to virtual machines, a modern server can typically run around 10 virtual machines or approximately 100 containers, resulting in a 10 times higher density of container-based mobile agents in the cloud. The migration time for a virtual machine is typically around 10 seconds, while a container can be migrated in about 1 second [17], making container migration approximately 10 times faster. Additionally, containers have the capability to call system functions of the operating system kernel running on the server. Hence, in principle, they can initiate container migration without the assistance of a separate daemon process operating on the host server.

Between 2002 and 2010, a period of experimentation occurred, during which two projects significantly advanced the field of virtualization containers in Linux. The VServer project involved patching the Linux kernel to partition it into virtual servers, resembling early versions of contemporary containers. The second project, OpenVZ, restructured the Linux kernel to enable the running of containers in production. Although successful, OpenVZ faced challenges in integrating its containerization technology into the standard Linux kernel, necessitating custom patches for functionality. Later, control groups and namespaces were introduced [18], providing the foundation for LXC containers to be incorporated directly into the stock Linux kernel. This development allowed the use of container-like functionalities without requiring any kernel patching.

During that period, Salomon Hykes was at the helm of dotCloud, a Platform as a Service (PaaS) company focused on adhering to standards in deploying distributed architectures for applications. They operated a cloud platform using LXC for three years, gaining significant operational experience. However, they realized that LXC was not ideal for their needs, leading them to develop a more stable and efficient tool that could facilitate container-based application deployment in large-scale hybrid cloud environments. This effort resulted in the creation of Docker; a widely popular technology based on the lib container format.

Docker containers cannot be live migrated between hosts—they can only be snapshotted and restored on the same or other host. The generally accepted method for managing Docker container data is to have stateless containers running in the production environment that store no data on their own and are purely transactional. Stateless containers store processed data on the outside, beyond the realm of their container space, preferably to a dedicated storage service that is backed by a reliable and available persistence backend. Another class of container instances are these that host storage services, like upon pattern is to use data containers. The runtime engines of these stateful services get linked at runtime with the data containers. In practical terms, this would mean having a database engine that would run on a container but using a “data container” that is mounted as a volume to store the state. Therefore, to run a cloud hosting environment, it is important to have a distributed storage solution, like Gluster and Ceph, to provide shared mount points. This is useful if the container instances move around the cloud based on availability.

Another popular container-based virtualization program for Windows and Linux operating systems is Parallels ® Virtuozzo. Virtuozzo permits live container migration, in contrast to Docker. The OpenVZ software, which is an open-source variant of this program, was used to produce the results reported in this work [11]. Only the Linux operating system is compatible with OpenVZ, and it uses a unique kernel. Numerous investigations on different container migration algorithm optimizations have been conducted [11]. Lazy migration and repeated migration are two of the better examples. Lazy migration refers to memory migration following container migration, in which memory pages are sent from the source server to the destination on demand. In the case of an iterative migration, memory migration occurs prior to container migration. We measured the migration time observed by the host T = 6.61 s and the migration time seen by the container = 2.25 s in our experiments with scaled-down OpenVZ containers with a size of 50 MB on a test system consisting of two nodes connected by a 100 Megabit Ethernet network. Because of the above-mentioned optimizations, the latter is three times smaller than the earlier.

By adding a system function that enables the container to request a host migration, we have modified the OpenVZ kernel. A container is added to a kernel queue by using this function; a special daemon reads this queue and migrates all the containers waiting there. Therefore, containers can only exit the host in a specific order. Our research shows that parallel migration is feasible, however the added performance is insignificant—migration speed only goes up by 8%. Additionally, only sequential migration helps to stabilize the swarm method, as demonstrated in our earlier study dealing with two hosts [19].

During the initial launch of our container, it lacks information about the host on which it was started. However, it can utilize an ICMP echo/reply mechanism to determine the IP address of the host. In this process, each ICMP packet carries a TTL (Time-to-Live) value, which is decremented as the packet is routed through routers. When the TTL value reaches 0, the packet is discarded, and an error ICMP packet is sent back. This error ICMP packet contains the IP address of the last router it passed through. To discover the IP address of the host, our container can send an ICMP echo packet with a TTL value of 1 to an arbitrary external IP address. Since the host system acts as a router for the container's network, this special packet will not reach its intended destination. However, the host system will respond by sending back an ICMP error packet containing its own IP address, allowing the container to detect the host's IP address.

All file systems mounted on the local file system of the host system are kept up-to-date. The file system of each container is accessible through a folder at /var/lib/vz/root/CID, where CID stands for a specific container identification number. Through a network file system like NFS, this location can be exported. Its IP address and export path (/var/lib/vz/root) must be known in order for our container to mount it locally. By counting the number of entries in the /var/lib/vz/root folder, our container can identify other containers running on the same host and determine their total count, denoted as N. Additionally, our container can utilize a custom system function, implemented by us, to check the number of containers (Q) queued for migration in the kernel. Alternatively, a container can log in via SSH to another host and obtain these parameters there. Each container is aware of the total number of hosts (H) and knows their respective IP addresses. It also possesses information about the total number of other containers (C) in the cloud, and these numbers can be updated dynamically during runtime by probing other containers and hosts using the ICMP echo/reply protocol, either by the container itself or by the host.

3. Swarm Algorithm

In various biological examples, intricate global behaviours emerge from simple interactions among numerous relatively unintelligent agents. Natural aggregation processes, such as nest construction, foraging, brood sorting, hunting, navigation, and emigration, rely solely on local interactions between individuals and their environment. For instance, ants demonstrate remarkable group behaviours through simple individual behaviours facilitated by physical contact and pheromone communication, resulting in colony-wide optimality. With advancements in technology, the development of cost-effective small robots equipped with sensors, actuators, and computation has become feasible. Swarm robotics, which employs large numbers of simple robots instead of a few sophisticated ones, offers significant advantages in terms of robustness and efficiency. Such systems can effectively handle various failures and unplanned behavior at the individual agent level without compromising task completion [20, 21–23, 24, 25]. These attributes make swarm algorithm an appealing solution for diverse problem domains.

This paper focuses on employing swarm intelligence for task scheduling in a complex distributed system—the cloud. By drawing inspiration from the self-organized behaviors found in nature and the potential of emerging robotics technologies, we aim to enhance the efficiency and adaptability of cloud-based task scheduling.

Now, based on pheromone robots [24,25], we develop a decentralized algorithm for container migration. The suggested method treats containers as mobile agents that have the ability to automatically repair themselves. This system is capable of quickly recovering from diverse agent death patterns and accommodating new agents without any problems at any place. The pheromone parameter "p," which is designated for each host, can either be repellent (0 p 1) or attractive (p 0). The whole procedure, which is carried out by a specific process running inside each container, can be summed up as follows:

- Use the method described in Sect. 2 to get the IP address of the host.
- Mount the /var/lib/vz/root folder exported by it.
- **loop**
- **repeat**
- Obtain the pheromone value *p* of the host.
- Generate a random number $0 < r < 1$.
- **until** $r < p$
- Randomly choose a host with an attractive pheromone $p < 0$.
- Ask the host to migrate to it.
- **repeat**
- Get the IP address of the host.
- **until** It's different from the previous one
  {Migration to another host is complete}
- Unmount /var/lib/vz/root from the old host.
- Mount it from the new host.
- **end loop**

Thus, the pheromone *p* can be viewed as a migration probability of a container. The simplest choice for *p* is the fraction of the number of containers on a host:

$$P = \frac{N-Q-n}{N-Q} \text{.................(1)}$$

above the equilibrium value where the containers are equally distributed between the hosts:

$$n = \frac{C}{H} \text{......................(2)}$$

Tasks are moved across nodes in the scenario up until n total tasks are present on each node. The nodes can be compared to gas containers in physics, and the processes running on them to gas molecules. Tubes connecting the containers serve as the network connections between nodes. Until it is the same in all containers, the gas pressure equalizes. The creation of self-repairing mobile agents uses a similar analogy [22]. It should be emphasized that Q must be eliminated from equation (1) in order to prevent container oscillations [19].

Practical cloud settings frequently have workloads with variable CPU and I/O loads. Therefore, attempting to distribute jobs equally among all available hosts may not always be the best course of action. The Dominant Resource Fairness (DRF) algorithm [26] can be used to determine the appropriate number of containers of a specific type (n) on each server as opposed to utilizing Eq. (2). DRF determines the required number of containers for every server separately. Consider a server with nine CPU cores and eight gigabytes of RAM, where container type A needs one core and four gigabytes of RAM and container type B needs three cores and one gigabyte of RAM. DRF would produce the appropriate numbers of containers for each, nA = 3 and nB = 2, respectively. Separate computations must be performed for each container type to derive the pheromone value (p) from Eq. (1). There are also different migration queues (Q) for different types of containers.

4. Methodology

Hardware and Software Configuration: The study used 18 machines equipped with an Intel® i5-3570 Quad-Core CPU and 8GB of RAM. These machines were connected through a dedicated 100 Megabit Ethernet network. All servers ran the OpenVZ software and the Debian GNU/Linux operating system. The Linux kernel was modified to incorporate additional system features.

Container Setup: A total of 306 identical containers were initially launched on 17 of the hosts, while the 18th host remained empty. Each container was around 100 MB in size and ran a Python script implementing the algorithm described in Section 3 of the study.

Algorithm Execution: The Python script within each container used the network scanning tool nmap to determine the number of containers and hosts on the network. It then calculated the mean number of containers (n) and periodically checked the pheromone value (p) to decide whether to migrate to another host. Each host ran a program that counted the number of containers on the filesystem and the number of containers waiting to be migrated. This data was accessible through a user process.

Data Collection: The number of containers (N) on each host was recorded over time, resulting in Figure 1. This graph showed the container distribution across hosts as they migrated.

5. Results and discussions

A total of 18 machines with an Intel® i5-3570 Quad-Core CPU and 8GB of RAM were used for the studies. A dedicated 100 Megabit Ethernet network was used to connect these servers. All of the servers used the OpenVZ software and the Debian GNU/Linux operating system. As explained in Section 3, the Linux kernel was altered by the addition of additional system features. At the outset, 306 identical containers were launched on the first 17 hosts, while the last host remained empty. This configuration is represented by the following notation:

$$N_i = 18,\ i = 1, \ldots 17,\ N_{18} = 0 \text{.................(3)}$$

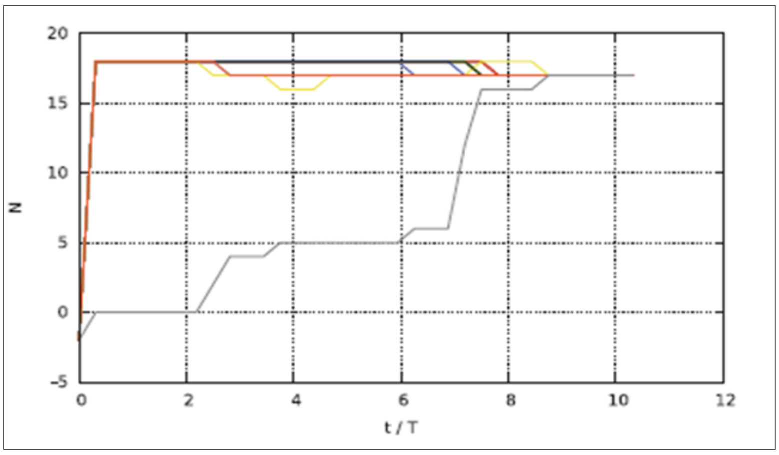

Figure 1 Number of containers on each host versus time

Each container had a size of approximately 100 MB, and the migration process to another host took around 16 seconds. Additionally, each container ran a Python script that implemented the algorithm outlined in Section 3. Initially, the script used nmap to scan the network and determine the number of containers (C), the number of hosts (H = 18), and their respective IP addresses. Subsequently, the mean number of containers (n) was calculated, and the script entered a loop in which it periodically checked the pheromone value (p). It then decided, based on the probability p, whether to migrate to another host. On each host server, a program was initiated that periodically (every 5 seconds) counted the number of containers (N) on the filesystem and the number of containers (Q) waiting to be migrated. Through the addition of a unique system function to the kernel, access to this data from a user process was made possible.

In Fig. 1 we have the numbers of containers N on each host plotted versus time t. It is seen from inspection of this plot that the containers can arrive to the destination host in parallel thus network bandwith was apparently not a problem during this experiment. Notice that around t _ 4 T the yellow line drops below the equilibrium value of N = 17—this happens because the migration process is inherently a probabilistic one. The migration probability is p = 1/18 but sometimes more than pN = 1 container can decide to jump to another host. Also, the containers do not move independently but interact with each other. If more than one excess container asks the host for migration, then one of them must wait in a queue until the first one leaves the host. The system reaches equilibrium and migration stops around t _ 9 T:

$$Ni = 17, i = 1, \ldots 18 \quad \text{……………….(4)}$$

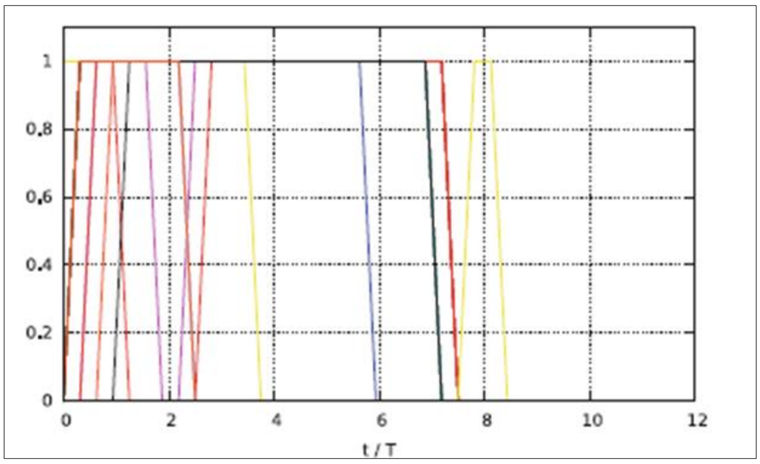

Figure 2 Number of containers waiting for migration on each host versus time. (Color figure online)

Thus, at average two containers arrive to the destination host during time T. Containers startup is not instantaneous. The experiment was arranged in such a way that the migration agent processes inside the containers were started in a loop—therefore some were started later than the other. In Fig. 2 we have the numbers of containers waiting for migration Q on each host plotted versus time t. Indeed, we see a small delay in entering the queue. The first container leaves the migration queue around t _ 1.5 T (red line) but is deleted from the file system with some delay only after $t >$ 2 T (c.f., Fig. 1).

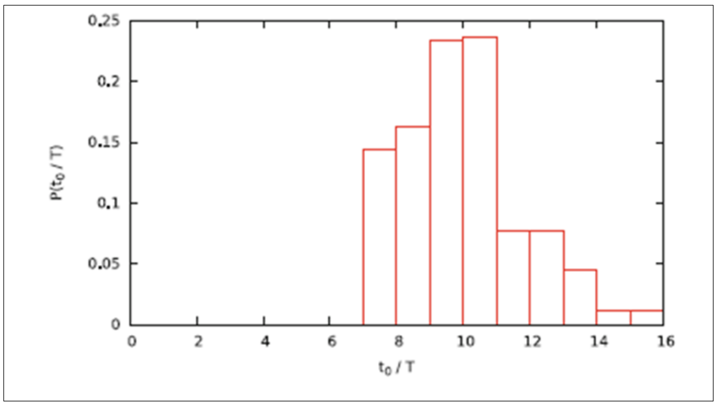

Figure 3 Histogram of times needed to reach equilibrium

To investigate the container self-organization process even further in Fig. 3 we have a histogram of times needed to reach equilibrium $t0$ obtained from 300 runs of the algorithm. It is seen that the case discussed earlier is a typical one.

6. Conclusion

In summary, the swarm of tasks in the cloud was implemented using the OpenVZ containerization software. Each task contains a mobile agent process responsible for managing its migration to other nodes within the cloud. The Contained Gas Model, known for self-repairing formations of autonomous robots, was adapted for this purpose. The tasks within the cloud autonomously organize themselves to maintain a balanced load among the servers. The system dynamically adjusts to the creation, destruction, and expansion of tasks, as well as the addition of new servers to the cloud. Furthermore, the system is designed to respond to server failures. By adding entries to its /var/lib/vz/root directory, a failing server can produce an artificial pheromone that will cause any jobs running on it to move away from it. Through testing on a simple cloud configuration with 18 nodes, the performance of the suggested swarm-like algorithm, which manages the containers, was assessed.

Compliance with ethical standards

Acknowledgments

The authors acknowledge the efforts of all the contributors to this research.

Disclosure of conflict of interest

The authors declare no conflict of interest.

References

[1] Balalaie, A., Heydarnoori, A., Jamshidi, P.: Migrating to cloud-native architectures using microservices: an experience report. In: Celesti, A., Leitner, P. (eds.) ESOCC Workshops 2015. CCIS, vol. 567, pp. 201–215. Springer, Heidelberg (2016). doi:10. 1007/978-3-319-33313-7 15.

[2] Savchenko, D., Radchenko, G., Taipale, O.: Microservices validation: mjolnirr platform case study. In: 2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), pp. 235–240. IEEE (2015)

[3] Th¨ones, J.: Microservices. IEEE Softw. 32(1), 116 (2015)

[4] Madhavapeddy, A., Scott, D.J.: Unikernels: rise of the virtual library operating system. Queue 11(11), 30 (2013)

[5] Calinciuc, A., Spoiala, C.C., Turcu, C.O., Filote, C.: Openstack and docker: building a high-performance iaas platform for interactive social media applications. In: 2016 International Conference on Development and Application Systems (DAS), pp. 287–290. IEEE (2016).

[6] Kratzke, N.: About microservices, containers and their underestimated impact on network performance. In: Proceedings of CLOUD COMPUTING 2015 (2015)

[7] Brewer, E.A.: Kubernetes and the path to cloud native. In: Proceedings of the Sixth ACM Symposium on Cloud Computing, pp. 167–167. ACM (2015)

[8] Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A.D., Katz, R.H., Shenker, S., Stoica, I.: Mesos: a platform for fine-grained resource sharing in the data center. In: NSDI, vol. 11, p. 22 (2011)

[9] Stubbs, J., Moreira, W., Dooley, R.: Distributed systems of microservices using docker and serfnode. In: 2015 7th International Workshop on Science Gateways (IWSG), pp. 34–39. IEEE (2015)

[10] Chafle, G.B., Chandra, S., Mann, V., Nanda, M.G.: Decentralized orchestration of composite web services. In: Proceedings of the 13th International World Wide Web Conference on Alternate Track Papers & Posters, pp. 134–143. ACM (2004)

[11] Mirkin, A., Kuznetsov, A., Kolyshkin, K.: Containers checkpointing and live migration. Proc. Linux Symp. 2, 85–90 (2008)

[12] Scheepers, M.J.: Virtualization and containerization of application infrastructure: a comparison. In: 21st Twente Student Conference on IT, pp. 1–7 (2014)

[13] Hacker, T.J., Romero, F., Nielsen, J.J.: Secure live migration of parallel applications using container-based virtual machines. Int. J. Space Based Situat. Comput. 12(1), 45–57 (2012)

[14] Aversa, R., Di Martino, B., Rak, M., Venticinque, S.: Cloud agency: a mobile agent-based cloud system. In: 2010 International Conference on Complex, Intelligent and Software Intensive Systems (CISIS), pp. 132–137. IEEE (2010)

[15] Haichun, N., Yong, L.: A mobile agent-based task seamless migration model for mobile cloud computing. In: 2014 IEEE Workshop on Advanced Research and Technology in Industry Applications (WARTIA), pp. 241–246. IEEE (2014)

[16] Thant, H.A., San, K.M., Tun, K.M.L., Naing, T.T., Thein, N.: Mobile agents-based load balancing method for parallel applications. In: APSITT 2005 Proceedings. 6th Asia-Pacific Symposium on Information and Telecommunication Technologies, pp. 77–82. IEEE (2005)

[17] Zhao, M., Figueiredo, R.J.: Experimental study of virtual machine migration in support of reservation of cluster resources. In: Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed Computing, p. 5. ACM (2007)

[18] Pike, R., Presotto, D., Thompson, K., Trickey, H., Winterbottom, P.: The use of name spaces in plan 9. In: Proceedings of the 5th Workshop on ACM SIGOPS European Workshop: Models and Paradigms for Distributed Systems Structuring, pp. 1–5. ACM (1992)

[19] Rusek, M., Dwornicki, G., Or_lowski, A.: Swarm of mobile virtualization containers. In: ´Swi _ atek, J., Borzemski, L., Grzech, A., Wilimowska, Z. (eds.) Information Systems Architecture and Technology: Proceedings of 36th International Conference on Information Systems Architecture and Technology – ISAT 2015 – Part III. AISC, vol. 431, pp. 75–85. Springer, Heidelberg (2016). doi:10.1007/978-3-319-28564-1 7

[20] Berman, S., Hal´asz, A., Kumar, V., Pratt, S.: Bio-inspired group behaviors for the deployment of a swarm of robots to multiple destinations. In: 2007 IEEE International Conference on Robotics and Automation, pp. 2318–2323. IEEE (2007)

[21] Cheah, C.C., Hou, S.P., Slotine, J.J.E.: Region-based shape control for a swarm of robots. Automatica 45(10), 2406–2411 (2009).

[22] Cheng, J., Cheng, W., Nagpal, R.: Robust and self-repairing formation control for swarms of mobile agents. In: AAAI, vol. 5, pp. 59–64 (2005)

[23] Dorigo, M., Trianni, V., S¸ahin, E., Groß, R., Labella, T.H., Baldassarre, G., Nolfi, S., Deneubourg, J.L., Mondada, F., Floreano, D., et al.: Evolving self-organizing behaviors for a swarm-bot. Auton. Robots 17(2–3), 223–245 (2004)

[24] Payton, D., Estkowski, R., Howard, M.: Progress in pheromone robotics. Intell. Auton. Syst. 7, 256–264 (2002)

[25] Payton, D., Estkowski, R., Howard, M.: Compound behaviors in pheromone robotics. Robot. Auton. Syst. 44(3), 229–240 (2003)

[26] Ghodsi, A., Zaharia, M., Hindman, B., Konwinski, A., Shenker, S., Stoica, I.: Dominant resource fairness: fair allocation of multiple resource types. In: NSDI, vol. 11, p. 24 (2011)

[27] Malavalli, D., Sathappan, S.: Scalable microservice based architecture for enabling DMTF profiles. In: 2015 11th International Conference on Network and Service Management (CNSM), pp. 428–432. IEEE (2015)

[28] Namiot, D., Sneps-Sneppe, M.: On micro-services architecture. Int. J. Open Inf. Technol. 2(9), 39 (2014)

[29] Pahl, C.: Containerisation and the paas cloud. IEEE Cloud Comput. 2(3), 24–31 (2015).